

# Compilers

## Topic 4

### The Symbol Table and Block Structure

Mick O'Donnell: [michael.odonnell@uam.es](mailto:michael.odonnell@uam.es)  
Alfonso Ortega: [alfonso.ortega@uam.es](mailto:alfonso.ortega@uam.es)



## Topic 1:

### The Symbol Table



## Symbol Tables (i)

### The Symbol Table

#### Motivation:

- Previously, we have seen how source code is analysed as a series of tokens (lexical analysis), and how these tokens are analysed as a structured program (syntactic analysis).
- These steps pay no regard to the variable names themselves, they only see each variable as a token “id”.
- Syntactic analysis checks identifiers are used appropriately WITHIN each statement (locally)
- Semantic analysis checks that identifiers are used appropriately within the program as a whole (globally)

```
int a;  
a = "hello";
```

Assigning  
wrong type

```
int a;  
b = 1;
```

Assigning  
undeclared  
variable

3

## Symbol Tables (i)

### The Symbol Table

#### Motivation:

- For semantic checking, we need to check whether:
  - Variable not declared multiple times
  - Variable declared before assigned
  - Variable assigned before referenced
  - Assignment compatible with declared type
  - Operations on variables compatible with type

```
int a;  
a = "hello";
```

Assigning  
wrong type

```
int a;  
b = 1;
```

Assigning  
undeclared  
variable

4

## Symbol Tables (i)

### The Symbol Table

- **Symbol Table :**
  1. **In a compiler:** a data structure used by the compiler to keep track of identifiers used in the source program. This is a compile-time data structure. Not used at run time.
  2. **In object files:** a symbol table (mapping var name to address) can be build into object programs, to be used during linking of different object programs to resolve reference.
  3. **In executables:** a symbol table (again mapping name to address) can be included in executables, to recover variable names during debugging.
- **Our Focus:** Symbol table in compiler

5

## Symbol Tables (i)

### The Symbol Table

#### Variable Declarations

- In static-typing programming languages, variables need to be declared before they are used. The declaration provides the data type of the variable.
  - E.g., `int a; float b; string c;`
- Most typically, declaration is valid for the scope in which it occurs:
  - **Function Scoping:** each variable is defined anywhere in the function in which it is defined, after the point of definition
  - **Block Scoping:** variables are only valid within the block of code in which it is defined, e.g,

```
prog xxx {int a; float b}
{
  int c;
  {
    int b;
    c = a + b;
  }
  return float(c) / b
}
```

6

## Symbol Tables (i)

### The Symbol Table

- **Identifiers:** User-supplied names, such as:
  - variable names,
  - function names,
  - labels (e.g., where *goto* is allowed)
- Symbol table typically implemented as a **hash table**:
  - KEY: the symbol
  - VALUE: information about the symbol

7

## Symbol Tables (i)

### The Symbol Table

#### Terminology:

- **Symbol** : the character string recognised during lexical analysis, e.g., "begin", "int", "A", ":", etc. are symbols.

```
begin
  int A;
  A = 100;
  A = A+A;
  output A
end
```

8

## Symbol Tables (i)

### The Symbol Table

#### Terminology:

- **Token**: the syntactic label for the symbol.
  - This label is that used during syntactic analysis. For example, assume we have a line of code:  

```
A = A + A ;
```
  - During lexical analysis, we label each symbol with its token:  

```
(id, 'A')(eqop, '=')(id, 'A')(plus, '+')(id, 'A')(semic, ';')
```
  - During Syntactic analysis, we might have rules:  

```
Statement := id eqop Expr semic  
Expr      := id plus id
```
  - We can see then that the tokens are simply the labels replacing symbols for syntactic parsing.
  - Tokens are the terminals of the syntactic analysis

9

## Symbol Tables (i)

### The Symbol Table

- **Types of Tokens**: tokens can be divided into:
  - Language-defined tokens:
    - **Reserved words**: sets of symbols defined by the language to have special meaning, e.g., “**begin**”, “**defun**”, etc.
    - **Operators**: =, +, \*, etc.
    - **Dividers**: { } ;
  - User-defined tokens:
    - **Identifiers**: names of variables, procedures, etc.
    - **Literals**: numbers, strings, etc. specified in the program.
- The symbol table is (in most cases) concerned only with storing **identifiers** and their attributes.

10

## Symbol Tables (i)

### The Symbol Table

- A simple symbol table:
  - **Data Structure:** A hash table where:
    - **Key:** a symbol
    - **Value:** the token for the symbol (id, num, etc.)

11

## Symbol Tables (i)

### The Symbol Table

- A simple symbol table:
  - **Data Structure:** A hash table where:
    - **Key:** a symbol
    - **Value:** the token for the symbol (id, num, etc.)
  - **Methods:**
    - `Insert (symbol, tok)` – set the token of *symbol* to *tok*
    - `Lookup (symbol)` – if *symbol* is known, return its tok, else return 0

12

## Symbol Tables (i)

### The Symbol Table

- A simple symbol table:
  - **Data Structure:** A hash table where:
    - Key: a symbol
    - Value: the token for the symbol (id, num, etc.)
  - **Methods:**
    - `Insert (symbol, tok)` – set the token of *symbol* to *tok*
    - `Lookup (symbol)` – if symbol is known, return its tok, else return 0
  - **Initialising for reserved words:** All reserved words are placed in the symbol table at the start, e.g.,
    - `Insert ("begin", begin)`This prevents reserved words being used as identifiers.

13

## Treatment of identifiers

### More complex Symbol Table

- In a more complex Symbol table, the “value” returned by the lookup is a record, containing various details about the symbol:
  - **The token:** id, num, etc.
  - **Usage:** variable, procedure, label
  - If a variable:
    - **Datatype:** integer, real, boolean, string, array, list, etc.
  - If a **structured object:** its structure
  - If a **procedure**,
    - its return datatype (if any),
    - Number of args, etc.
    - List of parameters, local variables, etc.
    - If it is recursive (calls itself)

14

## Treatment of identifiers

### Symbol Table

- The symbol table is mostly constructed during semantic analysis,
- Lexical analysis can record that a given token is an identifier
- But it is only after syntactic analysis that the compiler can know the context of the identifier:
  - In a variable declaration: `int A;`
  - In a variable assignment: `A = 54 / 3;`
  - In a variable reference: `if (A > 5) ...`
  - As a procedure name: `defun A { ... }`

15

## Treatment of identifiers

### Symbol Table: example

- Assume a language where
  - all variables must be declared before use
  - No global variables
  - No goto statements
  - We use a separate symbol table per function/method

16



### Symbol Table: example

- During semantic analysis, we advance through the program checking each identifier:
  - **Variable Declaration:**
    - Return **error** if not first declaration of variable
    - Else, add an entry for the variable:  
STATE=*uninitiated* TYPE= *type*
  - **Variable Assignment:**
    - Return **error** if variable not declared, or TYPE is illegal
    - Set the STATE to "initiated"
  - **Variable Reference:**
    - Return **error** if variable not declared, or TYPE is illegal
    - Return **error** if the variable is not initiated
  - **Procedure name:**
    - Add an entry for the procedure

17

### Symbol tables with block structures

#### Semantic Analysis: simple case

```
defproc myproc (int A, float B) {  
  int D, E;  
  D = 0;  
  E = A / round(B);  
  if (E > 5) {  
    print D  
  }  
}
```

Symbol Table

Symb	Token	Dtype	Init?
------	-------	-------	-------

18

## Symbol tables with block structures

### Semantic Analysis: simple case

```
defproc myproc (int A, float B) {  
    int D, E;  
    D = 0;  
    E = A / round(B);  
    if (E > 5) {  
        print D  
    }  
}
```

Symbol Table

Symb	Token	Dtype	Init?
myproc	id	procname	-

**Event:** identifier = procedure name

**Action:** Add name to symbol name

19

## Symbol tables with block structures

### Semantic Analysis: simple case

```
defproc myproc (int A, float B) {  
    int D, E;  
    D = 0;  
    E = A / round(B);  
    if (E > 5) {  
        print D  
    }  
}
```

Symbol Table

Symb	Token	Dtype	Init?
myproc	id	procname	-
A	id	int	yes

**Event:** identifier = variable declaration, function arg

**Action:** Add name to symbol name, as initialised

20

## Symbol tables with block structures

### Semantic Analysis: simple case

```
defproc myproc (int A, float E) {
  int D, E;
  D = 0;
  E = A / round(B);
  if (E > 5) {
    print D
  }
}
```

Symbol Table

Symb	Token	Dtype	Init?
myproc	id	procname	-
A	id	int	yes
B	id	float	yes

**Event:** identifier = variable declaration, function arg

**Action:** Add name to symbol name, as initialised

21

## Symbol tables with block structures

### Semantic Analysis: simple case

```
defproc myproc (int A, float B) {
  int D, E;
  D = 0;
  E = A / round(B);
  if (E > 5) {
    print D
  }
}
```

Symbol Table

Symb	Token	Dtype	Init?
myproc	id	procname	-
A	id	int	yes
B	id	float	yes

**Event:** identifier = variable declaration

**Check:** Already in symbol table? if so, fail

**Else:** Add name to symbol name, not initialised

22

## Symbol tables with block structures

### Semantic Analysis: simple case

```
defproc myproc (int A, float B) {  
  int D E;  
  D = 0;  
  E = A / round(B);  
  if (E > 5) {  
    print D  
  }  
}
```

Symbol Table

Symb	Token	Dtype	Init?
myproc	id	procname	-
A	id	int	yes
B	id	float	yes
D	id	int	no

**Event:** identifier = variable declaration

**Check:** Already in symbol table? if so, fail

**Else:** Add name to symbol name, not initialised

23

## Symbol tables with block structures

### Semantic Analysis: simple case

```
defproc myproc (int A, float B) {  
  int D, E;  
  D = 0;  
  E = A / round(B);  
  if (E > 5) {  
    print D  
  }  
}
```

Symbol Table

Symb	Token	Dtype	Init?
myproc	id	procname	-
A	id	int	yes
B	id	float	yes
D	id	int	no

**Event:** identifier = variable declaration

**Check:** Already in symbol table? if so, fail

**Else:** Add name to symbol name, not initialised

24

## Symbol tables with block structures

### Semantic Analysis: simple case

```
defproc myproc (int A, float B) {  
  int D, E;  
  D = 0;  
  E = A / round(B);  
  if (E > 5) {  
    print D  
  }  
}
```

Symbol Table

Symb	Token	Dtype	Init?
myproc	id	procname	-
A	id	int	yes
B	id	float	yes
D	id	int	no
E	id	int	no

**Event:** identifier = variable declaration

**Check:** Already in symbol table? if so, fail

**Else:** Add name to symbol name, not initialised

25

## Symbol tables with block structures

### Semantic Analysis: simple case

```
defproc myproc (int A, float B) {  
  int D, E;  
  D = 0;  
  E = A / round(B);  
  if (E > 5) {  
    print D  
  }  
}
```

Symbol Table

Symb	Token	Dtype	Init?
myproc	id	procname	-
A	id	int	yes
B	id	float	yes
D	id	int	no
E	id	int	no

**Event:** identifier = variable assignment

**Check:** ERROR if not in symbol table

**Action:** find entry in ST and set initialised

26

## Symbol tables with block structures

### Semantic Analysis: simple case

```
defproc myproc (int A, float B) {
  int D, E;
  D = 0;
  E = A / round(B);
  if (E > 5) {
    print D
  }
}
```

Symbol Table

Symb	Token	Dtype	Init?
myproc	id	procname	-
A	id	int	yes
B	id	float	yes
D	id	int	YES
E	id	int	YES

**Event:** identifier = variable assignment

**Action:** find entry in ST and set initialised

27

## Symbol tables with block structures

### Semantic Analysis: simple case

```
defproc myproc (int A, float B) {
  int D, E;
  D = 0;
  E = A / round(B);
  if (E > 5) {
    print D
  }
}
```

Symbol Table

Symb	Token	Dtype	Init?
myproc	id	procname	-
A	id	int	yes
B	id	float	yes
D	id	int	YES
E	id	int	YES

**Event:** identifier = variable reference

Check: report ERROR if not in symbol table

Check: report ERROR if not initialised

**Action:** none

28

## Symbol tables with block structures

### Problem case: fork in control

Where there is a possible fork in control, it may not be certain whether a variable is initialised or not.

For simple semantic analysis, at the point (1) we can know at compile time that the variable D might not be initialised, and issue a warning.

Better solutions to this problem in optimisation.

```
defproc myproc (bool A, int B)
  int D;
  if A: D = B;
                                     <----- (1)
  return D;
}
```